Generating Code from Hierarchical State-Based Requirements*

Mats P.E. Heimdahl
University of Minnesota, Institute of Technology
Department of Computer Science, 4-192 EE/CS Bldg.
Minneapolis, MN 55455
heimdahl@cs.umn.edu

David J. Keenan
Hughes Information Technology Systems
16800 E. Centretech Pkwy, Bldg. 485, M/S 5M-82
Denver, CO 80011-9046
dkeenan@redwood.dn.hac.com

1 Introduction

Computer software is playing an increasingly important role in safety-critical embedded computer systems, where incorrect operation of the software could lead to loss of life, substantial material or environmental damage, or large monetary losses. Although software is a powerful and flexible tool for industry, these very advantages have contributed to a corresponding increase in system complexity. It is becoming clear that the power software can bring to a system can also undermine the ability of the analyst to comprehend, and consequently control, the system's behavior.

In a previous investigation, the Irvine Safety Research Group, under the leadership of Dr. Nancy Leveson, developed a requirements specification language called the Requirements State Machine Language (RSML) suitable for the specification of safety-critical control embedded systems [14, 15]. To make RSML suitable as a requirements specification language usable by all stake holders in a specification effort, the syntax and semantics were developed with readability, understandability, and ease of use in mind. The usefulness of the language was demonstrated through the successful development of a requirements specification for a large commercial avionics system called TCAS II (Traffic alert and Collision Avoidance System II) [14, 15]. Furthermore, we have developed a collection of automated analysis procedures that check an RSML specification for desirable properties such as completeness, consistency, and determinism [10, 11].

However, even if a requirements specification is readable, understandable, and can be shown to be complete and consistent, designing and developing production quality code from such a black-box high-level specification can be a time consuming and error prone process.

To simplify and automate the design and implementation process, we have investigated the possibility of automatically generating code from RSML specifications. The semantics of RSML is relatively simple and is defined as a composition of simple mathematical functions defined by the state transitions in the model [11]. We have developed a tool that translates an RSML specification to executable code. The translation closely follows the formal semantics of RSML and, thus, makes verification of the correctness of the generated code simple. For the translations where this straightforward approach generates obviously inefficient code, we have defined easily verifiable automated correctness preserving optimizations.

Our goal is to generate production quality code directly from RSML specifications with as little human intervention as possible. To determine if this is a realistic goal and to get some early feedback on our approach, we have applied our translation technique to several small sample RSML specifications, including a part of the TCAS II requirements specification. Initial results show that the generated code is approximately 5-10 times slower and about twice as large as highly optimized hand generated code. These results show that automatic code generation is feasible, at least in the domain for which RSML was developed, and that our easily verifiable translation is a promising start.

A program transformation approach provides a means of keeping a specification and its implementation consistent. When a change is needed, the change is made in the high-level specification rather than in the source code. The code is then simply regenerated from the specification. Furthermore, provided the transformations preserve correctness, the generated code is guaranteed to correctly implement the specification and satisfy all properties that have been proven about the specification.

Generating executable code from some higher level notation has been an active research area since the development of the first compiler. Although a transformational approach does not solve the fundamental problems of complex system development, it allows us to work in a notation suitable for the problem domain, in our case RSML and safety-critical embedded control systems, and in that way eliminate many accidental problems associated with obscure notation and manual coding.

Many systems that transform specifications into executable code have been developed over the years. The following overview is by no means an exhaustive de-

*This paper has appeared in *Proceedings of the IEEE International Symposium on Requirements Engineering (RE'97)*, January, 1997.

scription of such systems, but rather a brief summary of some of the main transformational approaches.

Early transformational systems, such as the TI (Transformational Implementation) system [1], CIP (Computer-aided, Intuition-guided Programming) [2], and KIDS (Kestrel Interactive Development System) [16], were designed for use in general purpose programming. These types of systems are usually based upon wide spectrum languages that are iteratively transformed into lower level constructs. One of the major drawbacks to this approach is the need for the user to guide the transformational process and provide application-specific domain knowledge to the system. These systems can usually only perform a few simple transformations automatically, requiring the user to make decisions relating to complex data types and algorithms. The level of automation of such general-purpose systems varies, but it seems unlikely that a fully automatic transformational system can be built using this approach.

A different approach to code generation is taken by the Statemate system [6, 7, 12]. Statemate is a system for developing software for reactive systems that supports automatic generation of C, Ada, or VHDL code. A Statemate specification consists of three parts (1) a structural view defined by *module charts*, (2) a functional view defined by *activity charts*, and (3) a dynamic view defined by *Statecharts*.

The module charts are used to decompose a system into its main components. The functional decomposition of the components is captured using the activity charts. The modeling in Statemate is based on the stepwise refinement paradigm. The activity charts can be iteratively refined until atomic activities (or basic activities) can be described as simple transformations using a high-level programming language, for example, C or Ada. The activity charts fully capture the functionality of a system.

Statecharts are finite state machines augmented with hierarchy, parallelism, and modularity. In Statemate, the Statecharts are used to define the dynamic behavior of a system, that is, they determine when activities defined by the activity charts should be executed.

The transformation of a Statemate specification into code involves translating the activities to executable code. Activities with fully defined functionality will be directly translated to code, while activities with underspecified functionality will be translated into stubs that can be completed manually. The Statecharts are translated to a control program that determines when the code for representing the different activities will be invoked. The code generation is fully automated and results in what is considered to be prototype code.

The Statemate approach is similar to the RSML approach in some aspects. First, both approaches target embedded reactive systems and both are based on a state-based modeling approach. In fact, RSML is inspired by the Statecharts formalism and includes support for parallelism and hierarchical states.

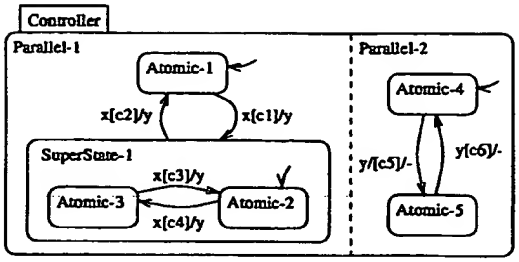

Figure 1: An example of a state machine.

Despite the similarities between Statecharts and RSML, the two modeling approaches are quite different. The main difference is that in RSML the full behavior of the system is captured using only the hierarchical state machine: RSML does not support activity charts and module charts. In the Statemate approach, the state machines are mainly used to control the scheduling of activities. Thus, the states reflect processing activity, that is, the states indicate which activities are currently executing. In RSML, on the other hand, the state machines are used to model the behavior of the physical components in a system: the states are used to visualize the system state.

During a large case study (TCAS) [3, 4, 15], we found that using the state machines to highlight the state of the system made it easy for the engineers (domain experts) such as avionics engineers, pilots, air frame manufacturers, and FAA representatives to understand and validate the requirements specification. The reason for this, we believe, is that an RSML specification is conceptually close to the application domain and becomes easier to validate. For a thorough discussion on this topic, the reader is referred to [15].

This difference in modeling approach leads to a different approach to code generation. While Statemate, for example, bases its code generation on stepwise refinement, we are generating all code directly from the state machines with fully automated transformations that can be proven correct.

The next section gives a brief description of the syntax and semantics of RSML. Sections 3 describes a provably correct mapping from RSML to executable code and Section 4 presents the results from our case study. Section 5 concludes.

2 The Semantics of RSML

RSML was developed as a requirements specification language for embedded systems. The language is based on hierarchical finite state machines and is in many ways similar to Statecharts by David Harel. For example, RSML supports parallelism, hierarchies, and guarded transitions borrowed from statecharts (Figure 1) [5, 8].

Transition(s): ESL-4 ⟶ ESL-2

Location: Own-Aircraft ▷ Effective-$SL_{s\text{-}30}$

Trigger Event: Auto-SL-Evaluated-$Event_{e\text{-}279}$

Condition:

			OR	
AND	Auto-$SL_{s\text{-}30}$ in state ASL-2	T	T	·
	Auto-$SL_{s\text{-}30}$ in one of {ASL-2,ASL-3,ASL-4,ASL-5,ASL-6,ASL-7}	·	·	T
	Lowest-$Ground_{f\text{-}241}$ = 2	·	·	T
	Mode-Selector = one of {TA/RA,TA-Only,3,4,5,6,7}	T	·	T
	Mode-$Selector_{v\text{-}34}$ = TA-Only	·	T	·

Output Action: Effective-SL-Evaluated-$Event_{e\text{-}279}$

Figure 2: A transition definition from TCAS II with the guarding condition expressed as an AND/OR table.

One of the main design goals of RSML was readability and understandability by non computer professionals such as, in our case, pilots, air frame manufacturers, and FAA representatives. During the TCAS project, we discovered that the guarding conditions required to accurately capture the requirements were often complex. The propositional logic notation traditionally used to define these conditions did not scale well to complex expressions and, thus, quickly became unreadable. To overcome this problem, we introduced a tabular notation for defining the guarding conditions (Figure 2). We call these tables AND/OR tables. The tables are read column-wise and were found to be very readable. To further increase the readability, we introduced many other syntactic conventions in RSML. For example, we allow expressions used in the predicates to be defined as mathematical functions (Other-Tracked-Relative-Alt-$Rate_{f\text{-}246}$), and familiar and frequently used conditions to be defined as macros (100-Ft-$Crossing_{m\text{-}195}$)[1]. A macro is simply a named AND/OR table defined elsewhere in the document. A detailed description of the full notation can be found in [15].

2.1 A Functional Framework

The behavior of a finite-state machine can be formally defined using a next-state relation. In RSML, this relation is modeled by transitions and the sequencing of events. Thus, one can view a graphical RSML specification as the definition of the mathematical next-state relation F.

Section 2.2 defines the static structure of the state

[1]The subscript is used to indicate the type of an identifier (f for functions, m for macros, and v for variables) and gives the page in the TCAS II requirements document where the identifier is defined.

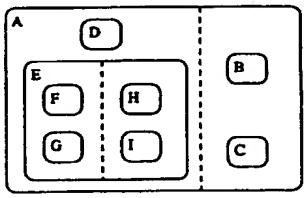

Figure 3: A sample state hierarchy

hierarchies. Although we are using a different notation, the basic ideas for the formalization of the hierarchical structure of the state machines is borrowed from Harel *et al.* [9]. Section 2.3 describes how the dynamic behavior defined by the transitions and events in RSML can be viewed as compositions of functions.

2.2 Hierarchical State Machines

An RSML state machine M can be described by a six-tuple: $M = (S, \leq, \sim, V, c_0, F)$ where:

S is a finite set of *states*.

$\leq$ is a tree-like partial ordering with a topmost point (called the *root*). This relation defines the hierarchy relation (or parent/child relation) on the states in S ($x \leq y$ meaning that x is a descendant of y, or x and y are equal).

In the graphical notation, this relation is visualized as containment (states are contained within superstates). In Figure 3, for example, B $\leq$ A, G $\leq$ A, I $\leq$ E, etc.

If the state x is a descendant of y ($x \leq y \wedge x \neq y$, denoted by $x < y$), and there is no z such that $x < z < y$, we say that the state x is a *child* of y (x *child* y).

Furthermore, we define $\sigma(y)$ as the set of all children of the state y, that is, $\sigma(y) = \{x \mid x \; child \; y\}$

$\sim$ is an equivalence relation on the states in $S - \{root\}$ that satisfies one additional property: whenever $x \sim y$, then x and y have the same parent.

$$x \sim y \Rightarrow \exists z \; : \; x, y \in \sigma(z)$$

The equivalence relation $\sim$ is used to partition the children of a state into disjoint sets called *parallel components*. In the graphical notation, this partition is indicated using dashed lines. In Figure 3, for example, the children of A are partitioned into two parallel components (equivalence classes) {B, C} and {D, E}.

V is a set containing the input and output histories of the model (the complete variable traces).

c_0 is the initial *global state* of the machine, $c_0 \in (Config \times V)$. A global state is an ordered pair consisting of a set of states, called the *configuration* of the machine, and a trace from V. The initial global state in Figure 3 is defined by the pair $(\{A, B, D\}, \emptyset)$.

F is a relation defining the global state changes in the machine M. F is a mapping $C \mapsto C$, where $C \subseteq (Config \times V)$. The relation F is also referred to as the *behavior* of M.

2.3 Next-State Mapping

The hierarchies and parallelism (defined by the relations $\leq$ and $\sim$), enforce a rigorous structure on the possible global states (the set C). The dynamic behavior (the possible global state changes) is defined by the next-state relation F $(C \mapsto C)$. In a model of a system with nontrivial functionality this mapping will be complex. However, the mapping can be viewed as a composition of smaller, less complex mappings. Specifically, F can be viewed as composed of simple *functions*.

In the graphical notation, these simple functions are defined by transitions. The *domain* of a function is defined by the source, that is, the state that the tail of the transition is leaving, and the guarding condition on the transition. The *image* of a function is defined by the destination of a transition, that is, the state the transition enters, and possible changes to variables. The functions represented by the transitions are then composed depending on the structure of the particular state machine being considered and the events defined on the transition. The semantics of RSML are defined using three basic compositions:

Union: The union composition of two functions $(g \cup h)$ merges the domains of the functions.

Serial: Serial composition $g(h(c))$ (or $g \circ h$) corresponds to normal functional composition.

Parallel: Parallel application is denoted $\langle h, g \rangle(x)$. Parallelism is modeled as interleaving, that is, an arbitrary ordering of serial compositions. The notation $\langle X \rangle$, where X is a set of functions, will be used to denote the parallel composition of the functions in X.

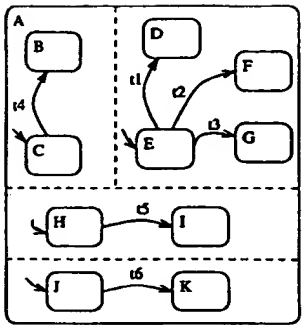

Figure 4: A sample state machine.

In RSML, union composition occurs between non-parallel transitions triggered by the same event. For example, the functions representing the transitions t_1, t_2, and t_3 in Figure 4 (assuming all are triggered by the same event) are composed in union.

Transitions triggered by the same event, but in parallel state components, are composed in parallel. In Figure 4, transitions t_3 and t_4 are composed in parallel (assuming they are triggered by the same event).

Finally, serial application is caused by the event propagation mechanism. Assume the transition t_5 is triggered by some external event and generates event e as an action. This event is picked up by transition t_6—that is, t_6 is triggered by e. Thus, transition t_5 is taken first and transition t_6 second. This sequencing is modeled as applying the functions representing t_5 and t_6 in series: $f_{t_6} \circ f_{t_5}(c)$.

A function f representing a transition can be textually defined by

$$f(c) = ((c.Conf - Q_s) \cup Q_d, \; v) \text{ if } (x \in c.Conf) \wedge p(c)$$

where Q_s and Q_d are sets of states, v is an updated variable trace, x is a state, and p is an arbitrary predicate over the global state c. The notation $c.Conf$ is used to refer to the *Configuration* part of the global state c. In the graphical representation, this function represents a transition starting in the state x and with the guarding condition p. If the transition is taken, the structure of the state machine may cause more states than x to be exited—for example, if x is a superstate. The set of states that is exited when the transition is taken is denoted by Q_s and the set that is entered by Q_d. Q_s and Q_d are the *source* ($Source(f)$) and *destination* ($Dest(f)$) of f respectively.

The functional definition of a complete RSML specification is recursively built from (composed of) the functional definitions of its components. To define this recursion we need to introduce some auxiliary concepts.

Let E be the set of all events in a model M. A transition is defined by a tuple $((C \mapsto C) \times E \times 2^E)$. The components of the tuple are denoted by *map*, *trigger*, and *actions* respectively. The *map* function is defined as outlined earlier in this section. Let T be the set

of all transitions. Furthermore, let $Stages = S \cup \Pi$, where Π is the set of equivalence classes of $\sim$. Also, for any $s \in S$, let $\pi(s)$ be the set of equivalence classes of children of s:

$$\pi(s) = \{x \in \Pi \mid x \subseteq \sigma(s)\}$$

For each $t \subseteq T$ we will define a function

$$g[t] \; : \; Stages \mapsto (C \mapsto C)$$

that defines the behavior of states and parallel components given a set of trigger events. The function g is defined by induction on $Stages$ over the relation $\ll$ defined as follows:
For all $s \in S, p \in \Pi$, and $st \in Stages$

$$s \ll st \quad \text{iff} \quad st \in \Pi \text{ and } s \in st \qquad (1)$$

$$p \ll st \quad \text{iff} \quad st \in S \text{ and } p \in \pi(st) \qquad (2)$$

Induction over $\ll$ is valid because it is well-founded: Whenever $s_1 \ll p \ll s_2$, it follows that $s_1 < s_2$. Therefore, $\ll$ does not contain an infinite descending chain.

The behavior of a composed state (a superstate) is defined as the parallel composition of its parallel state components. In Figure 4, for example, the behavior of the state A is defined as the parallel composition of its four parallel state components.

Definition 1 For any $p \in \Pi$ and $t \subseteq T$, the behavior (g) of a state $s \in S$:

$$g[t]_s = \langle \{g[t]_p \mid p \ll s\} \rangle$$

Informally, one can view the components of a composed state as processes, and the behavior of the composed state as the parallel execution of these processes.

The behavior of a set of states grouped in a parallel state component is defined as the union of (1) the behaviors of the states included in the component and (2) the behaviors introduced by the transitions between states at this level of abstraction. Let the notation $tr \sqsupset p$ (tr *belongs* to p) signify that a transition $tr \in T$ goes between two states in the parallel state component p. In Figure 4, the transition labeled with t_4 *belongs* to the parallel component $\{B, C\}$.

Definition 2 A transition tr belongs to the parallel component p of a state s, that is, $p \in \pi(s)$, (denoted by $tr \sqsupset p$) iff:

$$\exists x \in Source(tr.map) : x \in p$$

Definition 3 For any $s \in S$ and $t \subseteq T$, the behavior of a parallel state component $p \in \Pi$:

$$g[t]_p = \left(\bigcup_{m \ll p} g[t]_m \right) \cup \left(\bigcup_{tr \in t \wedge tr \sqsupset p} tr.map \right)$$

Informally, in a parallel state component, we will look for an applicable transition from the set of transitions between the states contained in this parallel component and among the transitions contained inside any of the states within the component.

Finally, the behavior of a model M under a specific event e can be defined. Let T_e be the set of all transition with the trigger $e \in E$, that is,

$$T_e = \{tr \in T \mid tr.trigger = e\}$$

Definition 4 The behavior of M under event $e \in E$ is defined as

$$F^e = g[T_e]_{root}$$

The rules defined above govern the behavior of M under one specific event, that is, all transitions in the model triggered by this one event are composed according to these rules. The behavior for all individual events in the model can now be modeled the same way. If an event e is generated, the function defined by the behavior under e (F^e), that is, the behavior generated by composing all transitions triggered by e, is applied, and a new system state is calculated. After a function has been applied and a new system state calculated, a new function is applied based on the output actions on the transitions used to construct the first function. We call the set of events generated as a result of output actions the *yield* of a next state calculation.

A sequence of function applications will follow. The next function is always determined by the yield of the previous function. Should the yield contain more than one event, the appropriate functions are composed in parallel. This sequence ultimately will stop when the next state calculation provides no yield.

3 Code Generation

The relative simplicity of the RSML semantics allows it to be translated into high-level programming constructs in a straightforward manner. In an attempt to make the generation of executable code easy to automate, easy to trace, and straightforward to verify, we have chosen to make a one-to-one mapping from the formal definition of the semantics to executable code. This translation provides a provably correct implementation of an RSML specification. However, such a direct approach introduces some inefficiencies in the generated code. Fortunately, these inefficiencies can be largely addressed through automated correctness preserving optimizations.

This section describes our translation approach and some optimizations are discussed in Section 4.

As mentioned in the previous section, the behavior of a single transition is defined by the function

$$f(c) = ((c.Conf - Q_s) \cup Q_d, v) \text{ if } (x \in c.Conf) \wedge p(c).$$

The behavior of this single transition can be trivially

```
bool ST()
{ if ((x ∈ c) and p){
    c = c − Q_s;
    c = c ∪ Q_d;
    v = modify(v);
    return TRUE; }
  return FALSE;
}
```

Figure 5: Pseudo-Code implementing a single transition.

implemented by the following program:

$$
\begin{array}{ll}
P: & \textbf{if} \\
 & ((x \in c.Conf) \wedge p(c)) \rightarrow I \\
 & \neg((x \in c.Conf) \wedge p(c)) \rightarrow \textbf{skip} \\
 & \textbf{fi} \\
 & \\
I: & c.Conf = c.Conf - Q_s; \\
 & c.Conf = c.Conf \cup Q_d; \\
 & v = modify(v);
\end{array}
$$

The correctness of the above program is easily verifiable by a weakest precondition proof.

The program in Figure 5 is the equivalent (pseudo)code for the program P, which executes a single transition upon occurrence of the trigger event. The return value will be used later during code generation to determine whether or not the transition was actually taken.

In the following discussion of composed and parallel states, the functions ge_1s and ge_1p correspond to the functions $g[t]_s$ and $g[t]_p$ defined in the previous section. Here, these functions are viewed as responding to event e_1, rather than executing transitions from set T which have trigger e_1. Since the set of transitions can be determined at translation time there is no need to pass them as a parameter. The applicable transitions can simply be included directly into the code.

The behavior of a composed state is defined as the parallel execution of its parallel components, where parallelism is modeled as arbitrary interleaving. Thus, if (p1, p2, ..., pn) are the parallel components of state s that can respond to event e, then the program in Figure 6 is one possible implementation of the behavior of composed state s (compare with Definition 1).

Parallelism is deterministic only if the behavior of the system is equivalent for all possible serial orderings. If this is not the case, the behavior is nondeterministic, and this implementation simply executes one of the possible behaviors.

The rules governing the behavior of a parallel component (union composition) can be modeled as a sequential attempt to find a transition that can be taken, that is, attempt to find a function that has the current state in its domain. By definition, a parallel component

```
void ge1s()
{ ge1p1();
  ge1p2();
  ...
  ge1pn();
}
```

Figure 6: Pseudo-Code implementing the behavior of a composed state under the event e_1.

```
void ge1p()
{ if (ST1())
    return ;
  if (ST2())
    return ;
  ...
  if (STn())
    return ;
  ge1s1();
  ge1s2();
  ...
  ge1sn();
}
```

Figure 7: Pseudo-Code implementing the behavior of a parallel component under the event e_1.

can successfully execute only one transition in response to a single event (Definition 3), and the program in Figure 7 enforces this behavior by immediately returning upon the successful completion of a single transition. (ST1, ST2, ... STn) are the corresponding functions for all transitions triggered by the event e_1 belonging to parallel component p, and (ge_1s1, ge_1s2, ... ge_1sn) are the functions defining the behavior (under the event e_1) of the the states that are children of p. Here the return value of the single transition function (ST in Figure 5) is used to exit the function upon successful execution of a transition. It should be noted that should there be nondeterminism, this implementation gives precedence to transitions at the highest level in the state hierarchy, which is not a requirement of the language.

As discussed earlier, the behavior of an entire machine under a specific event e_1 is simply ge_1s_{root}. Thus, there is one ge_1s_{root} function for each possible event e_1, and a function that responds to an arbitrary event e need only select the proper function ges_{root} to call. The program in Figure 8 implements this behavior, where (e_1, e_2, ..., e_n) are all events in the machine.

The event propagation (the sequence of function applications) can be modeled using two sets. One set (*toProcess*) to hold the events currently being evaluated and another set (*yield*) to collect the yield from

```
void execute(Event e)
{ switch (e) {
      case e1 : ge1sroot(); break;
      case e2 : ge2sroot(); break;
      ...
      case en : gensroot(); break; }
}
```

Figure 8: Pseudo-Code implementing the selection of function based on trigger event

```
set(Event) toProcess;
set(Event) yield;
void input(Event e)
{ toProcess.insert(e);
  bool moreEvents = TRUE;
  while (moreEvents) {
    for (each event e in toProcess) {
      execute(e); }

    if (yield.empty())
      moreEvents = FALSE;
    else{
      toProcess = yield;
      yield = emptySet; } }
}
```

Figure 9: Pseudo-Code implementing the sequence of transitions caused by the event-action semantics.

the transitions taken. When all events in *toProcess* have been processed, the events in *yield* are copied to *toProcess* and the a new evaluation can start. This cycle is repeated until we have an empty yield (compare the discussion of yield in Section 2). A function implementing this behavior can be seen in Figure 9.

Note also that the function implementing the behavior of the individual transitions must include code to put their actions into the yield set. Thus, the revised function for a single transition can be seen in Figure 10.

The code described in this section corresponds one-to-one with the formal definition of RSML. This simple implementation makes proofs of consistency between specification and implementation trivial.

To get a fully executable system, the only missing pieces are the input and output interfaces to the environment. The input and output mechanisms are often highly system dependent and usually not modeled in detail in a high-level specification. We are currently investigating how RSML can be augmented with a communications model that will allow us to generate code for hardware independent input and output modules. To make a fully executable system, the hardware

```
bool ST()
{ if(x ∈ c)
    if (p){
      c = c − Qs;
      c = c ∪ Qd;
      v = modify(v);
      yield.insert(actions);
      return TRUE; }
  return FALSE;
}
```

Figure 10: Modified code for a single transition

dependent code must be added separately. For now, however, we will just assume that modules exist for receiving messages from and sending messages to the environment. The input interfaces need simply receive a message from the environment, assign appropriate input variables, and pass the corresponding event to the input function to begin machine execution. Output interfaces are triggered by events generated while executing the machine. Thus, when an event is generated that is a trigger to an output interface, a call to the appropriate interface module can be made.

4 Case Studies

Two of the main impediments to widespread use of automatic code generation are code size and code efficiency: generated code is often unacceptably large and has poor run-time performance. The goal of our research is to develop a translation approach that will generate code of both acceptable size and acceptable performance. This is an ambitious goal. In many applications even using a compiler to translate from a high-level language such as C++ or Ada leads to unacceptably large and inefficient executables. Highly optimized hand coded assembly is still used in many time and space critical applications, for example, in the automotive industry [17]. Nevertheless, our goal is to be able to generate code that is efficient enough to be used in all but the most time and space critical applications.

To evaluate our approach and to get an indication of how automatically generated code compares to manually written code, we performed some small case studies.

We applied our technique to a subset of the TCAS II requirements specification [13, 15]. We translated a portion of the specification that determines the setting of a collection of aircraft parameters, for example, the effective sensitivity level (Effective-SL) that is a concept that determines how early to issue a collision warning to the pilot and Descend-Inhibit that determines if an aircraft is prohibited from descending (the state machine representing these parts of TCAS can be seen in Figure 11).

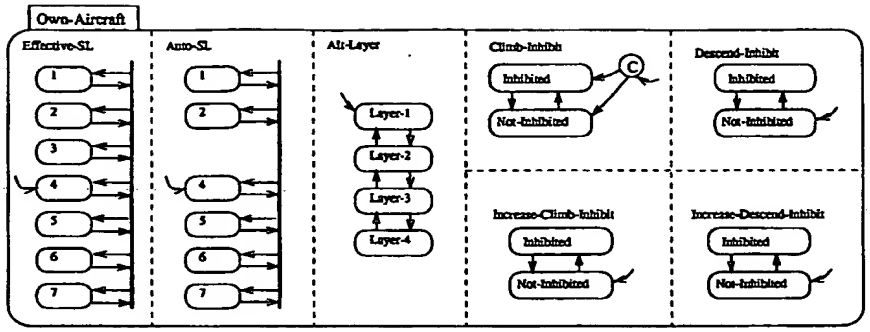

Figure 11: The state machine (Own-Aircraft) used in our case study.

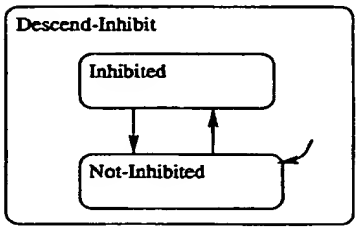

Figure 12: The state machine for Descend-Inhibit

Transition(s): Inhibited ⟶ Not-Inhibited

Location: Own-Aircraft ▷ Descend-Inhibit$_{s\text{-}30}$

Trigger Event: Surveillance-Complete-Event$_{e\text{-}279}$
Condition:

(Own-Tracked-Alt - Ground-Level) < 12,00ft | T

Output Action: Descend-Inhibit-Evaluated-Event$_{e\text{-}279}$

Figure 13: The transitions out of Inhibited

```
void Descend_Inhibit()
{ if (Inhib_to_NotInhib()) return;
  if (NotInhib_to_Inhib()) return;
}
```

Figure 14: The C++ code for the state machine Descend-Inhibit

```
bool Inhib_to_NotInhib()
{ if (isInConfig(DI_INHIBITED))
    if (OTrackAlt_Minus_GL_GT_1200()){
      removeFromConfig(DI_INHIBITED);
      addToConfig(DI_NOT_INHIBITED);
      Yield.insert(DESC_IN_EVAL_EVENT);
      return true;}
  return false;
}
```

Figure 15: The C++ code for the transition from Inhibited to Not-Inhibited

The pseudo-code outlined in the previous section can trivially be translated into executable code. In our case we chose C++, but any high-level language would be appropriate.

Consider the state machine for Descend-Inhibit in Figure 12 and the definition of the transition from the state Inhibited to the state Not-Inhibited (Figure 13). The C++ code corresponding to the state machine can be seen in Figure 14, and the code for the transition (or function) in Figure 13 is shown in Figure 15. The C++ code has a one-to-one correspondence to the pseudo-code in the previous section (compare the code with Figures 6 and 10).

We compared the efficiency and size of the automatically generated code with code derived from an English language and pseudo-code specification of the same system. The programs were compared on a set of test suites ranging from a few inputs to more than a million inputs.

While initial observations were somewhat disappointing (the generated code was approximately two times larger and approximately 30 times slower than the manually generated code), an analysis of the generated code uncovered several possibilities for automated optimizations which, when applied, significantly improved the efficiency of the generated code.

Optimizations: One of the first areas where the generated code demonstrates inefficiency is in the large

```
bool ST()
{ if (p){
    c = c − Q_s;
    c = c ∪ Q_d;
    v = modify(v);
    yield.insert(actions);
    return TRUE; }
  return FALSE;
}
```

Figure 16: Code for single transition with check for source state removed.

```
void gep()
{ if ((x1 ∈ c)&& ST1())
    return ;
  if ((x2 ∈ c)&& ST2())
    return ;
  ...
  if ((xn ∈ c)&& STn())
    return ;
  ges1();
  ges2();
  ...
  gesn();
}
```

Figure 17: Code for parallel component including check for source state.

number of function calls that must be made to find a transition that can be taken. The transitions triggered by an event are simply searched until an applicable transition is taken. A transition can only be taken, of course, if its source state is in the current configuration, and this test is done in the transition function (Figure 10). The first step in optimizing the code is thus to move this test for the source state's presence in the configuration into the calling function, the parallel component function (Figure 7). The resulting transition function and parallel component function can be seen in Figures 16 and 17.

This optimization allows the presence of a state in the configuration to be tested earlier and avoids a function call if the state is not in the configuration. The performance gain realized by this optimization is rather small, but it enables another optimization to be performed that will have a greater impact on the overall efficiency of the generated code.

The second optimization involves the repeated queries of the configuration to test for a state's presence. Looking at the modified code for a parallel component function (Figure 17), note that a query for the presence of a state in the configuration is made for

```
void gep()
{ switch (childActive(p)) {
  case x1 :
    if (ST1()) return;
    ges1();
    break;
  case x2 :
    if (ST2()) return;
    ges2();
    break;
  ...
  case xn :
    if (STn()) return;
    gesn();
    break; }
}
```

Figure 18: Further optimized code for parallel component.

each transition that is tested. If all queries could be reduced to one, a significant performance gain may be realized. This optimization was implemented by defining a function *childActive(p)* that returns the currently active child state of parallel component p. A parallel component can have only one active child state, and thus a single query of the configuration can determine which of the transitions may be applicable, that is, have their source states in the current configuration. The modified parallel component function can be seen in Figure 18.

The optimized generated code provided significantly better results than the unoptimized code. The optimized code was still approximately twice as large as the hand-generated code, but execution time was now only five times slower than the highly optimized hand-generated code.

The approach has also been applied to two additional small RSML specifications. In these cases the code was approximately 8 and 13 times slower than highly optimized hand-generated code. The size of the automatically generated code was about 10% larger than the hand-generated code in each case.

To summarize, the basis of our approach–the one-to-one mapping from specification to code–generated easily verifiable but inefficient code. However, by applying some obvious, fully automated, correctness preserving optimizing transformations the efficiency of the code can be significantly increased. By taking advantage of additional optimizations, for example, using the lazy evaluation of Boolean expressions in C++ when evaluating large guarding conditions, we believe the efficiency of the generated code can be comparable to efficient hand written code.

5 Summary and Conclusion

Experience has shown RSML to be an excellent tool for the modeling of safety-critical embedded control systems. However, without a means of formally verifying consistency between specification and implementation, there is no guarantee that the final implementation matches the specification. Automatic code generation offers the guarantee of correctness, as well as a significantly simplified and shortened development cycle, allowing the developers to concentrate on correctly specifying the system and validate its behavior rather than trying to overcome obscure implementation details.

Several other systems that generate code from specifications require the user to extensively guide the translation process, thus reducing the benefits of code generation. Because of the relative simplicity of the formal semantics of RSML and RSML's focus on a particular problem domain, fully automatic, provably correct code generation is possible.

In this paper, we outlined our approach for code generation from an RSML requirements specification to an implementation in C++. Our approach is based on a one-to-one mapping between the formal semantics of RSML and an implementation in a high-level language, in our case C++. The simplicity of the translation makes it trivial to verify its correctness. However, such a straightforward translation introduces inefficiencies in the generated code. To overcome this problem we apply fully automated, correctness preserving optimizing transformations that eliminate obvious inefficiencies.

To evaluate the feasibility of the approach, we compared the efficiency and size of the generated code with highly optimized hand written code. In this small case study the generated code was approximately one order of magnitude slower and twice as large as hand generated code. We are currently evaluating additional optimizations and we are convinced that it is possible for the generated code to achieve comparable performance to hand written code.

In addition to further applications of optimizations to the generated code, other future investigations will include the formulation of strategies to generate code for the remaining parts of RSML including input and output interfaces and timing constructs. Furthermore, more rigorous case studies of the performance of generated code will be conducted, including its performance on machines of various sizes and structures.

Acknowledgment: We would like to thank Martin Feather of JPL for his valuable comments on an earlier version of this paper.

References

[1] R. Balzer. A fifteen year perspective on automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1257–1267, November 1985.

[2] F. L. Bauer, B. Moller, M. Partsch, and P. Pepper. Formal program construction by transformations–computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, February 1989.

[3] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, vol-11(1):21–39, January 1994.

[4] S. Gerhart, D. Craigen, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995.

[5] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[6] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[7] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. Technical Report CS95-31, The Weizmann Institute of Science, October 1995.

[8] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.

[9] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts (extended abstract). In *2nd Symposium on Logic in Computer Science*, pages 54–64, Ithaca, NY, 1987.

[10] M. P.E. Heimdahl and N.G. Leveson. Completeness and Consistency Analysis of State-Based Requirements. Technical Report CPS-94-52, Michigan State University, October 1994. Accepted for publication in IEEE Transactions on Software Engineeriring.

[11] M. P.E. Heimdahl and N.G. Leveson. Completeness and Consistency Analysis of State-Based Requirements. *IEEE Transactions on Software Engineering*, TSE-22(6):363–377, June 1996.

[12] i Logix. The languages of Statemate, March 1987.

[13] N. G. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. TCAS II Requirements Specification.

[14] N. G. Leveson, M. P.E. Heimdahl, H. Hildreth, J. Reese, and R. Ortega. Experiences using Statecharts for a system requirements specification. In *Proceedings of the Sixth International Workshop on Software Specification and Design*, pages 31–41, 1991.

[15] N. G. Leveson, M. P.E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9), September 1994.

[16] D. R. Smith. Kids: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[17] Interview with D. Bogden. Reliability in the real world of embedded automotive software.